Hashing

Origins of the Term

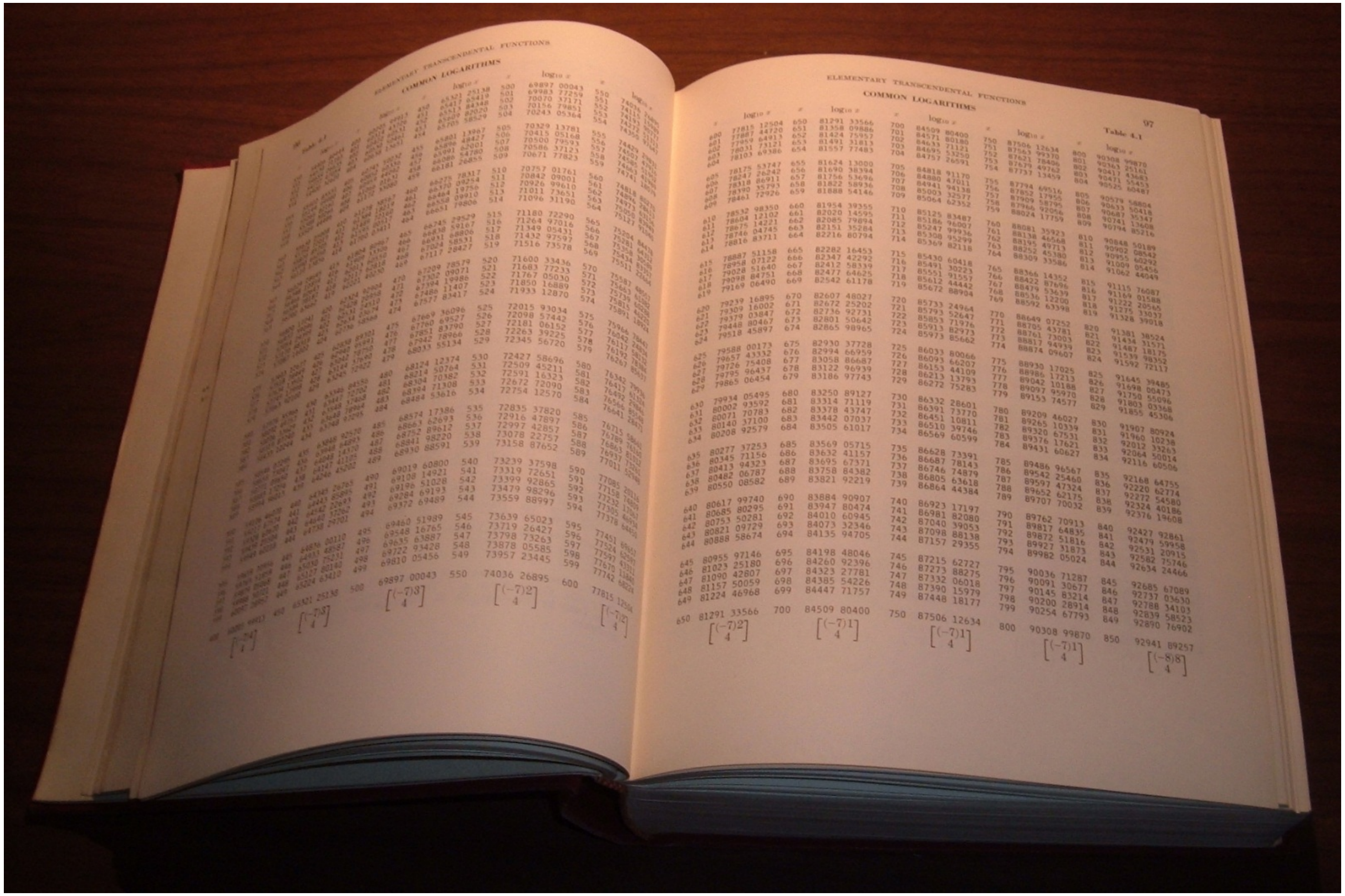
- The term "hash" comes by way of analogy with its standard meaning in the physical world, to "chop and mix." **D. Knuth** notes that **Hans Peter Luhn** of IBM appears to have been the first to use the concept, in a memo dated January 1953; the term hash came into use some ten years later.

Concept of Hashing

- In CS, a **hash table**, or a **hash map,** is a data structure that associates keys (names) with values (attributes).

  – Dictionary

Tables of logarithms



Basic Idea

- Use *hash function* to map keys into positions in a *hash table.*

Ideally

- If element *e* has key *k* and *h* is hash function, then *e* is stored in position *h(k)* of table.
- To search for *e*, compute *h(k)* to locate position. If no element, dictionary does not contain *e*.

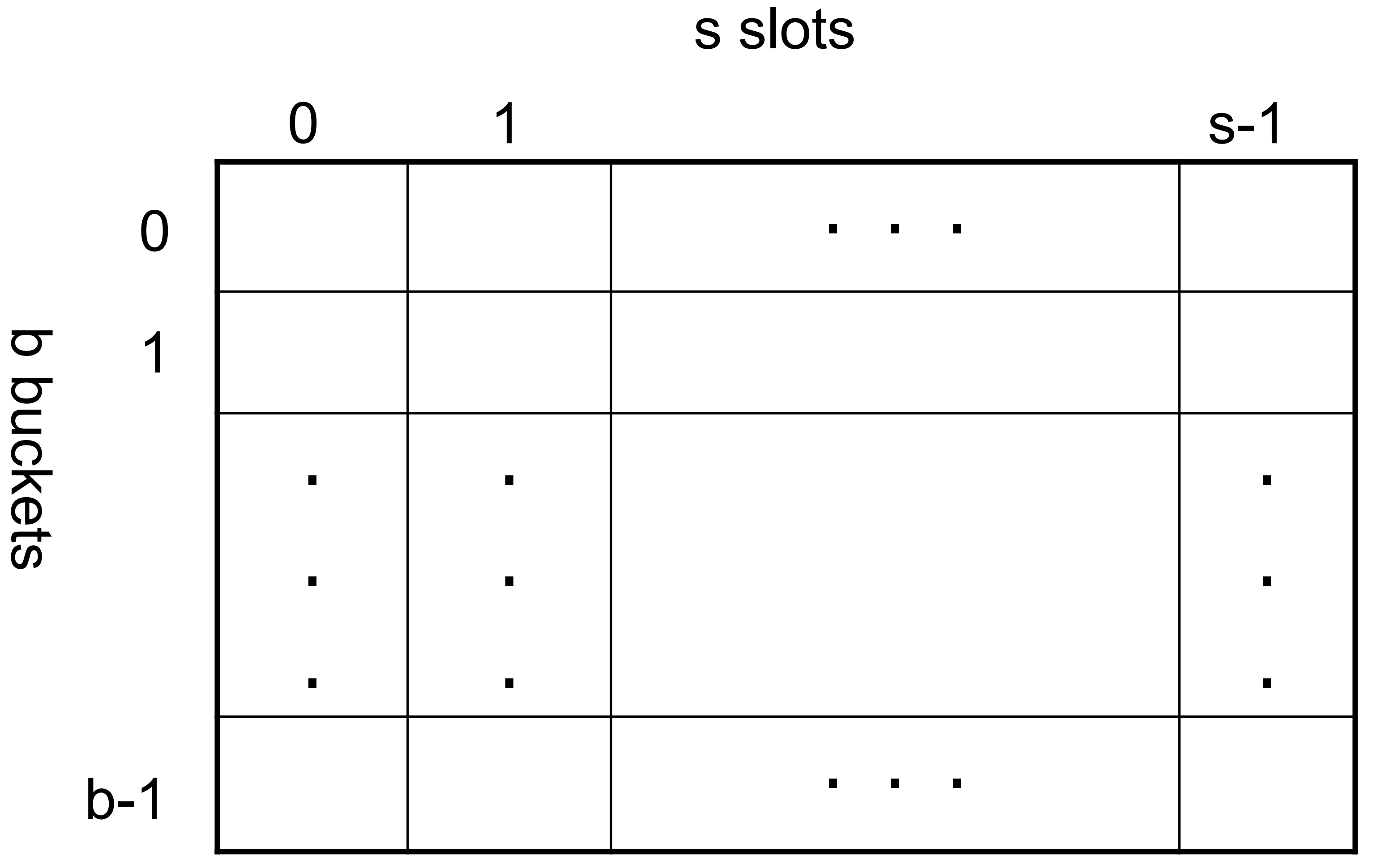
Search vs. Hashing

- Search tree methods: key comparisons
  - Time complexity: O(size) or O(log n)
- Hashing methods: hash functions
  - Expected time: O(1)
- Types
  - Static hashing
  - Dynamic hashing

Static Hashing

- Key-value pairs are stored in a fixed size table called a *hash table*.
  - A hash table is partitioned into many ***buckets***.
  - Each bucket has many ***slots***.
  - Each slot holds one record.
  - A hash function $h(k)$ transforms the identifier (key) into an address in the hash table

Hash table



Ideal Hashing

- Uses an array table[0:b-1].
  - Each position of this array is a bucket.
  - A bucket can normally hold only one slot.
- Uses a hash function f that converts each key k into an index in the range [0, b-1].
- Every pair (key, element) is stored in its home bucket table[f[key]].

Example

- Pairs are: (22,a), (33,c), (3,d), (73,e), (85,f).
- Hash table is table[0:7], b = 8.
- Hash function is key divide11.

(3,d)		(22,a)	(33,c)			(73,e)	(85,f)
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]

What Can Go Wrong?

(3,d)		(22,a)	(33,c)			(73,e)	(85,f)
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]

- Where does (26,g) go?
- Keys that have the same home bucket are synonyms.
  - 22 and 26 are synonyms with respect to the hash function that is in use.
- The bucket for (26,g) is already occupied.

Some Issues

- **Choice of hash function.**
  - ***Really tricky!***
  - To avoid collision (two different identifiers (keys) are hashed into the same bucket.)
  - Size (number of buckets) of hash table.
- **Overflow handling method.**
  - Overflow: there is no space in the bucket for the new identifier. (a new identifier or key is hashed into a full bucket.)

Example

synonyms:
char, ceil,
clock, ctime

overflow

	Slot 0	Slot 1
0	acos	atan
1		
2	char	ceil
3	define	
4	exp	
5	float	floor
6		
…		
25		

synonyms

synonyms

Criterion of Hash Table

- The loading density or loading factor of a hash table is $\alpha = n/(sb)$
  - s is the number of slots
  - b is the number of buckets
  - n is the number of keys in the table

Example

synonyms:
char, ceil,
clock, ctime

↑

overflow

	Slot 0	Slot 1
0	acos	atan
1		
2	char	ceil
3	define	
4	exp	
5	float	floor
6		
…		
25		

synonyms

synonyms

b=26, s=2, n=8, α=8/52=0.15, h(x)=the first char of x

Choice of Hash Function

- Requirements
  - easy to compute
  - minimal number of collisions
- A good hashing function distributes the key values uniformly throughout the range.

Some hash functions I

- Middle of square
  - H(x):= return middle digits of x^2
- Division
  - H(x):= return x % k
- Folding at the boundaries:
  - Partition the identifier x into several parts, and add the parts together to obtain the hash address
  - e.g. x=12320324111220; partition x into 123,203,241,112,20; then return the address 123+203+241+112+20=699

Shift at boundaries

- 12320324111220
- P1= 123 , P2= 203, P3= 241
  P4= 112 , P5= 20
- First reverse p2 and p4 to obtain 302 and 211, next add them to obtain

- 123+302+241+211+20=897

Some hash functions II

- Digit analysis:
  - If all the keys have been known in advance, then we could delete the digits of keys having the most skewed distributions, and use the rest digits as hash address.

Factors affecting the Performance of hashing

- The hash function
  - Ideally, it should distribute keys and entries evenly throughout the table
  - It should minimize *collisions*, where the position given by the hash function is already occupied
- The size of the table
  - Too big will waste memory; too small will increase collisions and may eventually force *rehashing* (copying into a larger table)
  - Should be appropriate for the hash function used – and a prime number is best
- The collision resolution strategy
  - *Separate chaining*: chain together several keys/entries in each position
  - *Open addressing*: store the key/entry in a different position

Choosing the table size to minimize collisions

- As the number of elements in the table increases, the likelihood of a *collision* increases - so make the table **as large as practical**
- If the table size is 100, and all the hashed keys are divisible by 10, there will be many collisions!
  - Particularly bad if table size is a power of a small integer such as 2 or 1

Collisions may still happen, so we need a *collision resolution strategy*

Collision resolution: open addressing (1)

Probing: If the table position given by the hashed key is already occupied, increase the position by some amount, until an empty position is found

- **Linear probing**: increase by 1 each time [mod table size!]
- **Quadratic probing**: to the original position, add 1, 4, 9, 16,…

Use the collision resolution strategy when inserting *and* when finding (ensure that the search key and the found keys match)

With open addressing, the table size should be double the expected no. of elements

Linear Probing – Get And Insert

- divisor = b (number of buckets) = 17.
- Home bucket = key % 17.

0				4				8				12				16
34	0	45				6	23	7			28	12	29	11	30	33

- Insert pairs whose keys are 6, 12, 34, 29, 28, 11, 23, 7, 0, 33, 30, 45

Linear Probing – Delete

0				4				8				12				16
34	0	45				6	23	7			28	12	29	11	30	33

- Delete(0)

0				4				8				12				16
34		45				6	23	7			28	12	29	11	30	33

- Search cluster for pair (if any) to fill vacated bucket.

0				4				8				12				16
34	45					6	23	7			28	12	29	11	30	33

Linear Probing – Delete(34)

0				4				8				12				16
34	0	45				6	23	7			28	12	29	11	30	33

0				4				8				12				16
	0	45				6	23	7			28	12	29	11	30	33

- Search cluster for pair (if any) to fill vacated bucket.

0				4				8				12				16
0		45				6	23	7			28	12	29	11	30	33

0				4				8				12				16
0	45					6	23	7			28	12	29	11	30	33

Linear Probing – Delete(29)

0				4				8				12				16
34	0	45				6	23	7			28	12	29	11	30	33

0				4				8				12				16
34	0	45				6	23	7			28	12		11	30	33

- Search cluster for pair (if any) to fill vacated bucket.

0				4				8				12				16
34	0	45				6	23	7			28	12	11		30	33

0				4				8				12				16
34	0	45				6	23	7			28	12	11	30		33

0				4				8				12				16
34	0					6	23	7			28	12	11	30	45	33

Quadratic Probing

- Linear probing searches buckets $(H(x)+i)\%b$
- Quadratic probing uses a quadratic function of *i* as the increment
- Examine buckets $H(x)$, $(H(x)+i^2)\%b$, $(H(x)-i^2)\%b$, for $1<=i<=(b-1)/2$
- b is a prime number of the form $4j+3$, j is an integer

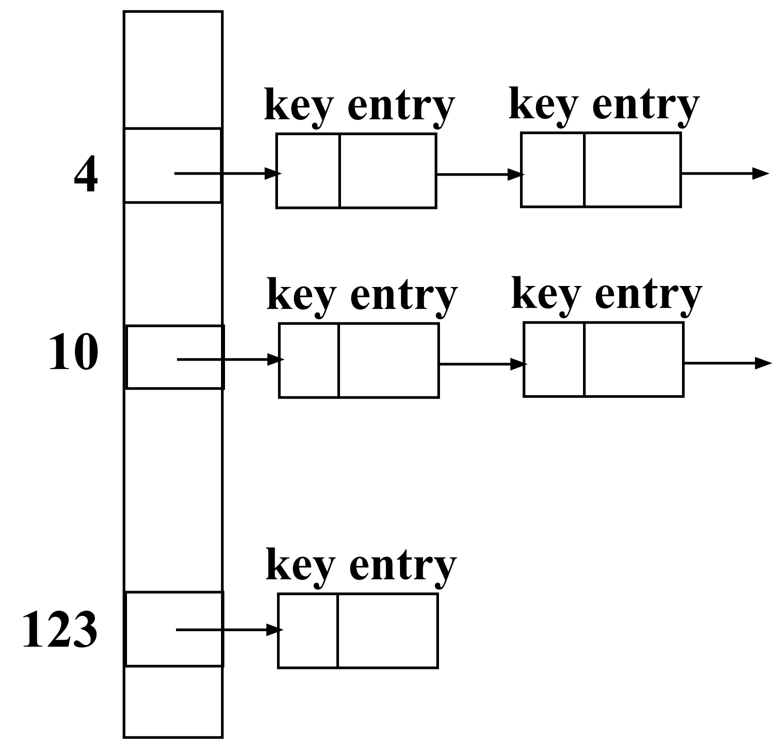
Quadratic Probing

- **Quadratic probing** is a solution to the clustering problem
- However, whereas linear probing guarantees that all empty positions will be examined if necessary, quadratic probing does not
  - e.g. Table size 16 and original hashed key 3 gives the sequence: 3, 4, 7, 12, 3, 12, 7, 4…
- More generally, with quadratic probing, insertion may be <u>impossible</u> if the table is more than half-full!
  - Need to *rehash* (see later)

Collision resolution: chaining

- Each table position is a linked list
- Add the keys and entries anywhere in the list (front easiest)
- Advantages over open addressing:
  - Simpler insertion and removal
  - Array size is not a limitation (but should still minimise collisions: make table size roughly equal to expected number of keys and entries)
- Disadvantage
  - Memory overhead is large if entries are small

No need to change position!



Rehashing: enlarging the table

- To *rehash*:
  - Create a new table of double the size (adjusting until it is again prime)
  - Transfer the entries in the old table to the new table, by re-computing their positions (using the hash function)
- When should we rehash?
  - When the table is completely full
  - With quadratic probing, when the table is half-full or insertion fails
- Why double the size?
  - If n is the number of elements in the table, there must have been $n/2$ insertions before the previous rehash (if rehashing done when table full)
  - So by making the table size $2n$, a constant cost is added to each insertion

Applications of Hashing

- Compilers use hash tables to keep track of declared variables
- A hash table can be used for on-line spelling checkers — if misspelling detection (rather than correction) is important, an entire dictionary can be hashed and words checked in constant time
- Game playing programs use hash tables to store seen positions, thereby saving computation time if the position is encountered again
- Hash functions can be used to quickly check for inequality — if two elements hash to different values they must be different
- Storing sparse data